# A Framework for Elastic Execution of Existing MPI Programs

Thesis

Presented in Partial Fulfillment of the Requirements for the Degree
Master of Science in the Graduate School of The Ohio State University

By

Aarthi Raveendran, B.S.

Graduate Program in Computer Science and Engineering

The Ohio State University

2011

Thesis Committee:

Dr. Gagan Agrawal, Advisor

Dr. Christopher Stewart

# ABSTRACT

There is a clear trend towards using cloud resources in the scientific or the HPC community, with a key attraction of cloud being the *elasticity* it offers. In executing HPC applications on a cloud environment, it will clearly be desirable to exploit elasticity of cloud environments, and increase or decrease the number of instances an application is executed on during the execution of the application, to meet time and/or cost constraints. Unfortunately, HPC applications have almost always been designed to use a fixed n umber of resources.

This work focuses on the goal of making existing MPI applications elastic for a cloud framework. Considering the limitations of the MPI implementations currently available, we support adaptation by terminating one execution and restarting a new program on a different number of instances. The components of the system include a decision layer which considers time and cost constraints, a framework for modifying MPI programs, and a cloud-based runtime support that can enable redistributing of saved data, and support automated resource allocation and application restart on a different number of nodes.

Using two MPI applications, the feasibility of our approach is demonstrated, it is shown that outputting, redistributing, and reading back data can be a reasonable approach for making existing MPI applications elastic. The decision layer with a feedback model is designed to monitor the application by interact with it at regular

intervals, and perform scaling with the assistance of resource allocator when necessary. This approach is tested using the same two applications and is used to meet the user demands of maximum specified input time or budget.

I dedicate this work to my friends

# ACKNOWLEDGMENTS

Foremost, I would like to express my sincere gratitude to my advisor Prof. Gagan Agrawal for the continuous support of my Masters study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me through my research and writing of this thesis.

My sincere thanks also goes to my thesis committee member Dr. Christopher Stewart, for his encouragement and insightful comments.

I would like to thank my fellow labmate Tekin Bicer, for providing invaluable assistance and help during the course of my Masters.

My deepest gratitude goes to my family for their unflagging love and support throughout my life; this dissertation could have been simply impossible without them.

# VITA

| | |
|---|---|
| 2005 | B.E., Electronics and Communication Engineering,<br>Anna University,<br>Chennai, India. |
| 2009-2011 | Masters Student,<br>Department of Computer Science and Engineering,<br>The Ohio State University. |

# PUBLICATIONS

### Research Publications

Aarthi Raveendran, Tekin Bicer, Gagan Agrawal "A Framework for Elastic Execution of Existing MPI Programs". *25th IEEE International Parallel & Distributed Processing Symposium, May 2011*

# FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

High Performance Computing, Grid Technologies Prof. Gagan Agrawal

# TABLE OF CONTENTS

**Page**

# LIST OF FIGURES

**Figure** **Page**

# CHAPTER 1

# INTRODUCTION

## 1.1 HPC Applications on Cloud

Scientific computing has traditionally been performed using resources on supercomputing centers and/or various local clusters maintained by organizations. However, in the last 1-2 years, cloud or utility model of computation has been gaining momentum rapidly. Besides its appeal in the commercial sector, there is a clear trend towards using cloud resources in the scientific or the HPC community.

The notion of on-demand resources supported by cloud computing has already prompted many users to begin adopting the Cloud for large-scale projects, including medical imaging [15], astronomy [3], BOINC applications [7], and remote sensing [9], among many others. Many efforts have conducted cost and performance studies of using Cloud environments for scientific or data-intensive applications. For instance, Deelman *et al.* reported the cost of utilizing cloud resources to support a representative workflow application, Montage [3]. Palankar *et al.* conducted an in-depth study on using S3 for supporting large-scale computing[12]. In another work, Kondo *et al.* compared cost-effectiveness of AWS against volunteer grids [7]. Weissman and Ramakrishnan discussed deploying Cloud proxies [16] for accelerating web services.

Multiple cloud providers are now specifically targeting HPC users and applications. Though initial configurations offered by the cloud providers were not very suited for traditional *tightly-coupled* HPC applications (typically because they did not use high performance interconnects), this has been changing recently. In November 2010, Mellanox and Beijing Computing Center have announced a public cloud which will be based on 40 Gb/s Infiniband. Amazon, probably the single largest cloud service provider today, announced *Cluster Compute Instances* for HPC in July 2010. This allows up to a factor of 10 better network performance as compared to a regular collection of EC2 instances, and an overall application speedup of 8.5 on a "comprehensive benchmark suite"[1].

The key attractions of cloud include the *pay-as-you-go* model and *elasticity*. Thus, clouds allow the users to instantly scale their resource consumption up or down according to the demand or the desired response time. Particularly, the ability to increase the resource consumption comes without the cost of *over-provisioning*, i.e., having to purchase and maintain a larger set of resources than those needed most of the time, which is the case for the traditional *in-house* resources. While the elasticity offered by the clouds can be beneficial for many applications and use-scenarios, it also imposes significant challenges in the development of applications or services. Some recent efforts have specifically focused on exploiting the elasticity of Clouds for different services, including a transactional data store [2], data-intensive web services [8], and a cache that accelerates data-intensive applications [1], and for execution of a bag of tasks [10].

[1]Please see www.hpcinthecloud.com/offthewire/Amazon-Introduces-Cluster-Compute-Instances-for-HPC-on-EC2-98321019.html

While executing HPC applications on a cloud environment, it will clearly be desirable to exploit elasticity of cloud environments, and increase or decrease the number of instances an application is executed on during the execution of the application. For a very long running application, a user may want to increase the number of instances to try and reduce the completion time of the application. Another factor could be the resource cost. If an application is not scaling in a linear or close to linear fashion, and if the user is flexible with respect to the completion time, the number of instances can be reduced, resulting in lower $nodes \times hours$, and thus a lower cost.

## 1.2 Specific Goals

Unfortunately, HPC applications have almost always been designed to use a fixed number of resources, and cannot exploit elasticity. Most parallel applications today have been developed using the Message Passing Interface (MPI). MPI versions 1.x did not have any support for changing the number of processes during the execution. While this changed with MPI version 2.0, this feature is not yet supported by many of the available MPI implementations. Moreover, significant effort is needed to manually change the process group, and redistribute the data to effectively use a different number of processes. Thus, existing MPI program are not designed to vary the number of processes. Adaptive MPI [5] can allow flexible load balancing across different number of nodes, but requires modification to the original MPI programs, and can incur substantial overheads when load balancing is not needed. Other existing mechanisms for making data parallel programs adaptive also do not apply to existing MPI programs [4, 17]. Similarly, the existing cloud resource provisioning frameworks cannot help in elastic execution of MPI programs [6, 11, 13].

## 1.3 Contributions

This thesis describes the work towards the goal of making existing MPI applications elastic for a cloud framework. Considering the limitations of the MPI implementations currently available, adaptation is supported by terminating one execution and restarting a new program on a different number of instances. To enable this, a decision layer constantly monitors the progress of the application and the communication overheads. This decision layer can terminate the application at certain points (typically, at the end of an iteration of the outer time-step loop) during which the live variables are collected and redistributed. These live variables are read by the restarted application, which continues from the iteration at which it was terminated.

The components of this system include:

- An automated framework for deciding when the number of instances for execution should be scaled up or down, based on high-level considerations like a user-desired completion time or budget, monitoring of the progress of the application and communication overheads.

- A framework for modifying MPI programs to be elastic and adaptable. This is done by finding certain points in the program where interation with the decision layer has to be done, and where variables identified to be live can be output to allow a restart from a different number of nodes if necessary. The long-term goal is to develop a simple source-to-source transformation program to generate the version with these capabilities.

- A cloud-based runtime support that can enable redistributing of saved data, and support for automated resource allocation and application restart on a different number of nodes.

## 1.4 Background: Amazon Cloud

This section gives some background on the Amazon Web Services (AWS), on which this work has been performed.

AWS offers many options for on-demand computing as a part of their Elastic Compute Cloud (EC2) service. EC2 nodes (*instances*) are virtual machines that can launch snapshots of systems, i.e., *images*. These images can be deployed onto various *instance types* (the underlying virtualized architecture) with varying costs depending on the instance type's capabilities.

For example, a Small EC2 Instance (`m1.small`), according to AWS[2] at the time of writing, contains 1.7 GB memory, 1 virtual core (equivalent to a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor), and 160 GB disk storage. AWS also states that the Small Instance has *moderate* network I/O. In stark contrast, an Extra Large EC2 instance (`m1.xlarge`) contains 15 GB memory, 4 virtual cores with 2 EC2 Compute Units each, 1.69 TB disk storage with *high* I/O. Many other such instance types exist, also with varying costs and capabilities.

Amazon's persistent storage framework, Simple Storage Service (S3), provides a key-value store with simple ftp-style API: `put`, `get`, `del`, etc. Typically, the unique keys are represented by a filename, and the values are themselves the data objects, i.e., files. While the objects themselves are limited to 5 GB, the number of objects that can be stored in S3 is unrestricted. Aside from the simple API, the S3 architecture has been designed to be highly reliable and available. It is furthermore very inexpensive to store data on S3.

[2]AWS Instance Types, http://aws.amazon.com/ec2/instance-types

Another feature of AWS is the availability of compute instances with three options: *on-demand instances*, *reserved instances*, and *spot instances*. An *on-demand* instance can be acquired and paid for hourly. Organizations can also make a one-time payment for *reserved instances*, and then receive a discount on the hourly use of that instance. With *spot instances*, Amazon makes its unused capacity, at any time, available through a lower (but variable) price.

## 1.5 Organization

This work describes the runtime framework developed for the Amazon EC2 environment to implement this overall idea. The former part of the thesis demonstrates the feasibility of this approach using two MPI applications, and it is shown that the monitoring framework has a negligible overhead, and outputting, redistributing, and reading back data for adapting application can be a reasonable approach for making existing MPI applications elastic. The latter part of the thesis validates framework with the results obtained for user contraints of time and cost and indicates that the system is able to meet the expectations of the user.

# CHAPTER 2

# FRAMEWORK DESIGN

This section describes the dynamic resource allocation framework by giving a simple illustration of the actual design. The overall goals of the framework are first described followed by an explanation of the necessary modifications on the source code for allowing elastic execution. This is followed by a description of the functionality of the various modules of the framework in detail.

## 2.1 Overall Goals

As stated earlier, we are driven by the observation that parallel applications are most often implemented using MPI and are designed to use a fixed number of processes during the execution. This is a crucial problem considering the driving features of cloud services, i.e., elasticity and the pay-as-you-go model.

The problems addressed with this framework can be categorized as *dynamic resource (de)allocation* and *data distribution* among the reallocated resources during the execution. The two constraints that can be specified by the user are considered. The user defined constraints are either based on a specific *time frame* within which the user would want the application to complete, or based on a *threshold value of the cost* that they are willing to spend. Clearly, it is possible that the execution cannot

be finished within the specified time or the cost. Thus, these constraints are supposed to be *soft* and not *hard*, i.e, the system makes the best effort to meet the constraints. The system is also being designed to be capable of providing feedback to the user on its ability to meet these constraints.

These goals are accomplished with several runtime support modules and making modifications to the application source code. In the long-term, we expect these source code modifications to be automated through a simple source-to-source transformation tool.

## 2.2 Execution Model and Modifications to the Source Code

Here, a simple illustration of the idea of decision making is depicted. This framework specifically assumes that the target HPC application is iterative in nature, i.e., it has a *time-step loop* and the amount of work done in each iteration is approximately the same. This is a reasonable assumption for most MPI programs, so this does not limit the applicability of our framework. This assumption, however, has two important consequences. First, the start of each (or every few) iteration(s) of the time-step loop becomes a convenient point for monitoring of the progress of the application. Second, because we only consider redistribution in between iterations of the time-step loop, we can significantly decrease the *check-pointing* and *redistribution* overhead. Particularly, a general check-pointing scheme will not only be very expensive, it also does not allow redistribution of the data to restart with a different number of nodes.

Figure 2.1 shows how the source code can be modified to handle monitoring of an iterative application, by adding a small piece of code at end of a particular number of

```
Input:  monitor_iter, iter_time_req,
        curr_iter, range
Output: true if solution converges, false otherwise

{ * Initiate MPI * }
data := read_data();
t0 := current_time();
While curr_iter<MAX_ITER Do
  { * Call update module * }
  rhonew := resid(data);
  If rhonew<EPS Then
      return true;
  Endif
  If (curr_iter%monitor_iter) = 0 and curr_iter ≠ 0 Then
      t1 := current_time();
      avg_time := (t1-t0)/curr_iter;
      If avg_time>(iter_time_req + range) Then
          { * Store data to a file * }
          { * Inform decision layer and expand resources * }
      Else If avg_time<(iter_time_req − range) Then
          { * Store data to a file * }
          { * Inform decision layer and shrink resources * }
      Endif
  Endif
  curr_iter := curr_iter+1;
Endwhile
return false;
```

Figure 2.1: Simple illustration of the idea

time step loops. The modification to the source code of an iterative application and implementation of the decision logic is shown. The *monitoring interval*, the *required iteration time*, *current iteration*, and the *range* are taken as runtime parameters. The monitoring interval determines the number of iterations after which the monitoring has to be done.

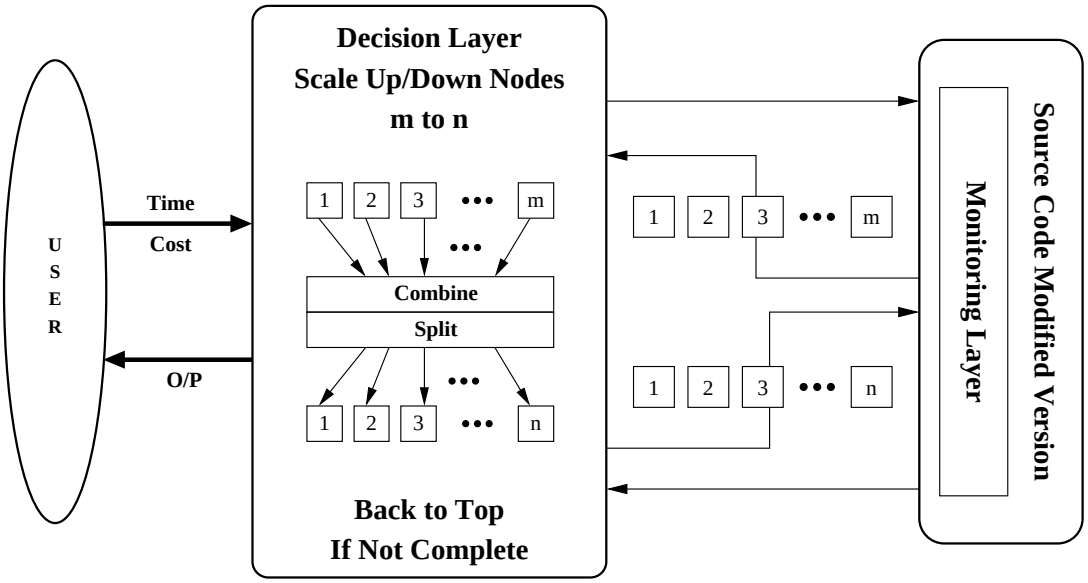


Figure 2.2: Execution Flow

This approach assumes that the user has a fixed execution time in mind, and the system can allocate more nodes to finish the execution if needed. In chapter 3 , it is shown that the framework can consider other constraints, such as the need to minimize the execution cost while staying within a time-limit. In such a case, this framework can reduce the number of nodes, provided the predicted execution time will stay within the limit.

In Fig. 2.1, the required iteration time is calculated by the decision layer based on the user's input. This value is used for checking the progress of the program based on the average iteration time. At any point, if there is a necessity to stop and restart the program on a new number of nodes, it is important to know how many iterations have already been completed and from which point the new set of nodes have to continue. This is given as one of the inputs, *curr_iter*, to the program. The

main iteration is thus started from curr_iter. The value of this variable in the first run is zero. It is also important to make sure that reallocation of the processing nodes is not done so frequently, otherwise the overhead of restart of the nodes and redistribution of the data might not be tolerable. A control parameter called *range* in this code, is given as another input to the program. Hence, every time the average iteration time is compared with the required iteration time, it is checked if the former falls within a range around the latter. Based on the deviation from the range, the decision to change the number of nodes is made. For each iteration, after processing the matrix, the application checks the computation time if the monitoring interval has been completed. If the average iteration time is above the required range, it means that the progress is not good enough and hence scaling up of the number of nodes will be necessary. On the contrary, if it is below the range, the application can afford to run slower and instance cost can be cut down by deallocating some of the nodes as the progress is much better than expected.

In the case where scaling has to be done, a decision is made and so the data needs to be redistributed among a new number of processes. The data which is distributed among the current set of processes needs to be collected at the master node and redistributed to the new set of nodes.

It is important to note here that not all data involved in the program needs to be carried over to the subsequent iterations. Only the *live* variables (and arrays) at the start of a new iteration need to be stored and redistributed. Furthermore, if the array is read-only, i.e. it has not been modified since the start of the execution, it does not need to be stored back before terminating one execution. Instead, the original dataset can be redistributed and loaded while restarting with a different set of nodes.

In this elastic framework, each processes stores a portion of the array that needs to be collected and redistributed to a file in the local directory. Other components of our framework are informed of the decision of the monitoring layer to expand or shrink the resources. The application returns *true* if the solution is converged, so that the decision layer does not restart it again. In case, the solution is not converged, *false* is returned which indicates that restarting and redistribution are necessary. The application is terminated and the master node collects the data files from the worker nodes and combines them . After launching the new nodes or deallocating the extra nodes based on the decision made by the monitoring layer, the decision layer in the master node splits the data and redistributes it to the new set of nodes.

The application is started again and all the nodes read the local data portions of the live arrays that were redistributed by the decision layer. The main loop is continued from this point and the monitoring layer again measures the average iteration time and makes a decision during the monitoring interval. If the need of restarting does not arise and the desired iteration time is reached, then the application continues running . Otherwise, the same procedure of writing the live data to local machines, copying them to master node and restarting the processes are repeated. Figure 2.2 depicts the execution flow of the system.

This approach is based on a static decision layer model with no feedback, where the required iteration time is assumed to be fixed. Chapter 3 shows how this model is made dynamic by moving the monitoring layer from the application souce code to the outer decision layer. As stated above, the main goal is to develop a simple source-to-source transformation tool which can automatically modify the MPI source code. The major steps in the transformation will be: 1) identifying the time-step loop,

which is typically the outer-most loop over the most compute-intensive components of the program, 2) finding live variables or arrays at the start of each iteration of the time-step loop, and finding the read-only variables, and 3) finding the distribution of the data used in the program.

## 2.3 Runtime Support Modules

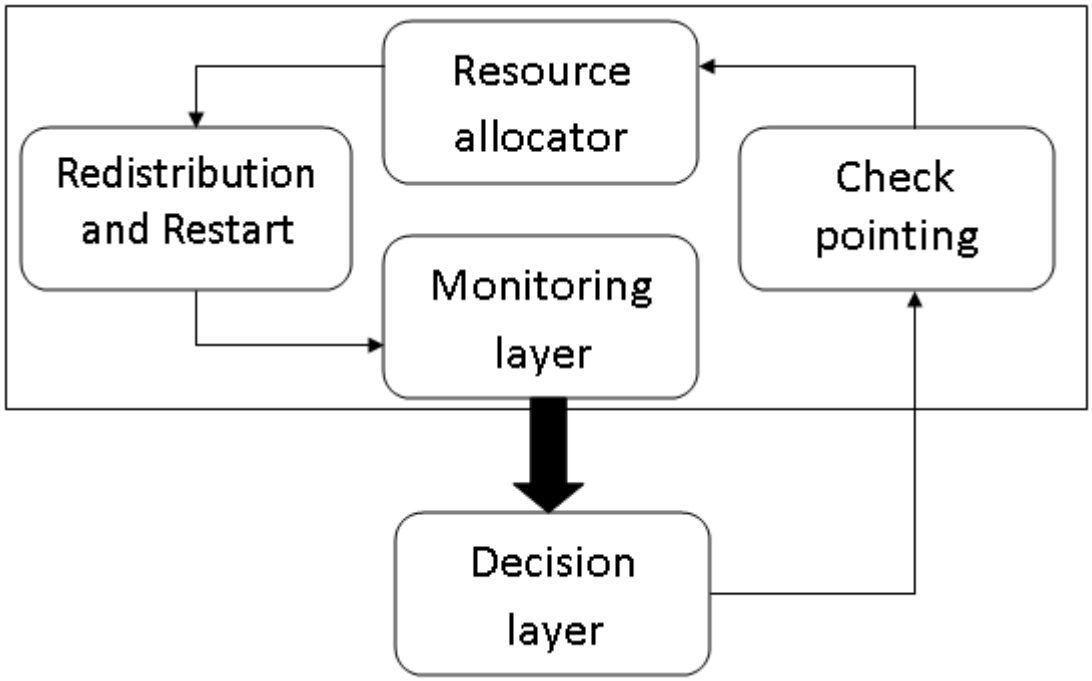


Figure 2.3: Components of Our Framework

The various components of the framework are now described. The interaction between these components is shown in Figure 2.3. Also, note that the role of the monitoring layer has already been explained above, so we do not elaborate it any further. The monitoring layer keeps interacting with the decision layer, which initiates

a checkpointing, to be followed by resource allocation, and then redistribution and restart. The decision layer interacts with the user to get the inputs and preferences of time and cost. It also interacts with the application program on monitoring the progress and deciding if a redistribution is needed. Most of the underlying logic has been explained in the previous subsection. The decision layer will be explained in detail in Chapter 3 along with the improved feedback model. This model also taking into consideration, the communication costs incurred by the application, in addition to the per iteration cost, in order to allow a better prediction of the execution time of the application while using fewer or more nodes.

### 2.3.1 Resource Allocator

One of the ways by which this framework enables elastic execution is by transparently allocating (or deallocating) resources in the AWS environment, and configuring them for the execution of the MPI program. New instances are requested and monitored by resource allocator until they are ready. For execution of the program, an MPI cluster needs to be set-up. The MPI implementation used is *MPICH2* and the process manager employed is *mpd* (multipurpose daemon) that provides both fast startup of parallel jobs and a flexible run-time environment that supports parallel libraries. The mpd daemon needs to be started so that it can spawn the mpich processes. The main advantage of mpd is that they can start a job on several nodes in less than a second. It also has fault tolerance - it can start jobs and deliver signals even in the face of node failure. For the mpd job launcher to work correctly, each of the nodes has to open connection with two of its nearest neighbors and hence all the nodes should form a *communication ring*. The order of this ring does not matter. A

Python script is used for this MPI configuration process. It runs commands to get the current state of the instances, their host-names and the external domain names. A *hosts* file containing a list of host-names has to be present in every node, as this will help mpd to initiate the execution among the processes. The nodes are configured for a password-less login from the master node and the host file is copied over from the master node to the slave nodes. Then, mpd is booted on the master node and all the other nodes join the ring. Once this is completed, the MPI environment is set and ready to run parallel jobs on multiple instances. The binary code has to be present on all nodes before execution and is hence copied from the master node to the other nodes by the resource allocator layer. The application can now be launched by giving the required iteration time as a runtime parameter.

In case a reallocation or deallocation is needed, the data portions residing on the nodes need to be collected at the master node, combined together, and then redivided into the new number of nodes. This is illustrated in Figure 2.2 where the number of nodes is being scaled from $m$ to $n$. The redivided data is then transferred to the new group of instances, which read the data and continue working on them from the point where they were terminated. Thus the whole process is repeated until the program finishes its execution.

### 2.3.2 Check-pointing and Data Redistribution Module

Data collection and redistribution depend on the type of application and the type of data. Multiple design options were considered for this, in view of the support available on AWS. Amazon S3 is a storage service provided by Amazon that can be accessed by the EC2 instances. For arrays that are not modified at each iteration

can be stored in small sized chunks in S3. Later, during a node launch, each of the nodes can download the chunks of data required by them and continue with their computation. This design is very efficient for unaltered data as it saves the overhead of writing to and reading from a file. For variables that are modified in each iteration, file writes and reads are used to write and read the data. The remote file copy command, *scp*, is used to transfer files to the master node, and again for copying the new pieces of each node.

Combining and redistributing data require the knowledge of how the original dataset was divided among the processes (e.g.: row-wise, column-wise, two-dimensional etc.). Currently, this information is provided as an annotation to the framework. In the future, our source-to-source transformation tool can be used to automatically extract this information. Based on the data distribution information and the number of initial and final instances, the redistribution module can generate the portions of the dataset each node will need.

Note that the current design performs aggregation and redistribution of data centrally on a single node. This can easily be a bottleneck if the initial and/or the final number of instances is quite large. In the future, we will implement more decentralized array redistribution schemes [14].

## 2.4 Applications and Experimental Results

In this section, the approach and framework performance is evaluated with 2 applications. The feasibility of our approach is demonstrated and the performance of the runtime modules we have implemented has been shown.

The experiments were conducted using 4, 8, and 16 Amazon EC2 small instances. The processing nodes communicate using MPICH2. The framework was evaluated with two MPI applications: *Jacobi* and *Conjugate Gradient* (CG). Jacobi is a widely used iterative method which calculates the eigenvectors and eigenvalues of a symmetric matrix. Since the Jacobi application processes and manipulates the matrix in each iteration, the updated matrix needs to be collected and redistributed among the compute nodes in the case of adaptation. The data redistribution is done using parallel data transfer calls, and the overhead of the data transmission time is significantly reduced. The NAS CG is a benchmark application that calculates the largest eigenvalue of a sparse, symmetric, positive definite matrix, using the inverse iteration method. A specific number of outer iterations is used for finding the eigenvalue estimates and the linear system is solved in every outer iteration. The dominant part of the CG benchmark is a matrix vector multiplication. The matrix is a sparse one and stored in compressed row format. This matrix is not manipulated during the program execution, thus it can be stored in a shared storage unit, i.e. the Amazon S3. The matrix is divided into chunks, and these chunks can be distributed, retrieved, and processed by the allocated EC2 instances.

The matrix processed for our Jacobi execution had 9K $\times$ 9K double values ($\sim$618 MB). This matrix needs to be collected and redistributed in the case of compute

instance reallocation. For CG, the matrix has 150K × 150K double values. However, only the vector needs to be redistributed, and its size is 1.14 MB.

Table 2.1: Jacobi application without scaling the resources

| No. Nodes | W/O Redist. (sec) | W/ Redist. (sec) | MPI Config. (sec) | Data Movement (sec) | Overhead (%) |
|---|---|---|---|---|---|
| 4 | 2810 | 2850 | 71 | 3 | 1 |
| 8 | 1649 | 1720 | 89 | 2.5 | 4 |
| 16 | 1001 | 1087 | 87 | 3.6 | 9 |

The first experiment involved executing Jacobi for 1000 iterations, and redistributing once (after 500 iterations). To be able to evaluate the redistribution and restart overhead, we "redistributed" the execution with the same number of nodes. This version included overheads of MPI configuration, data redistribution, copying of the source files to all nodes, and the actual program restart. Table 2.1 presents the execution times and the major overheads associated with redistribution of data, particularly, the MPI configuration and data movement costs. The overheads of this system range from 0.01% to 0.1% and show small increments with increasing number of compute instances. The main reason for the overhead is due to the MPI configuration time. It consists of collecting the host information of the newly initiated computing instances and preparing the configuration file that lets MPI daemon set up the MPI groups. This process can introduce small overheads, however the larger computation times are expected to further dominate these times. The parallel data

redistribution effectively transfers the updated data, and minimizes the data transmission time.

Some reasonable speedups are also observed with increasing number of instances. Since the communication tends to be relatively slow between Amazon EC2 instances, the speedups are not close to linear. As stated above, redistribution is done once between 1000 iterations. The overheads will be relatively higher if the redistribution is done more often. But, even if the redistribution was done after every 50 iterations, based on the above experiments, the overheads will still be less than 2%.

Table 2.2: Jacobi application with scaling the resources

| Starting Nodes | Final Nodes | MPI Config. (sec) | Data Movement (sec) | Total (sec) | Overhead (%) |
|---|---|---|---|---|---|
| 4 | 8 | 81 | 3 | 2301 | 3 |
| 4 | 16 | 84 | 3 | 1998 | 5 |
| 8 | 4 | 80 | 3 | 2267 | 2 |
| 8 | 16 | 95 | 3.8 | 1386 | 4 |
| 16 | 4 | 99 | 3.5 | 2004 | 5 |
| 16 | 8 | 97 | 3 | 1390 | 5 |

Table 2.2 shows how the runtime modules perform when the system actually scales up and down. The *Overhead* column now shows the *estimated overheads*, considering a projected execution time derived from the related *W/O Redist.* columns of Table 2.1. As we saw in the previous experiments, the overheads stay very low (0.02-0.05% overhead for redistributing once in 1000 iterations). The MPI configuration is the dominating factor for the overall overhead and it increases while the numbers of final nodes increase for the same starting nodes configuration.

Table 2.3: CG application without scaling the resources

| No. Nodes | W/O Redist. (sec) | W/ Redist. (sec) | MPI Config. (sec) | Data Movement (sec) | Overhead (%) |
|---|---|---|---|---|---|
| 4 | 834 | 879 | 25 | 2.5 | 5 |
| 8 | 997 | 980 | 58 | 3 | 0 |
| 16 | 1030 | 1105 | 96 | 2.7 | 7 |

The same set of experiments were reported with CG and the results are presented in Tables 2.3 and 2.4. Redistribution was now performed once during 75 iterations of the application. The first observation from these results is that unlike Jacobi, the performance of CG does not improve with an increasing number of nodes. This is because CG is more communication-intensive. It is expected that improvements in communication performance in clouds will help speedup an application like CG in the future (Amazon Cluster Compute Instance apparently has improved performance, although this has not been experimented with so far).

Table 2.4: CG application with scaling the resources

| Starting Nodes | Final Nodes | MPI Config. (sec) | Data Movement (sec) | Total (sec) | Overhead (%) |
|---|---|---|---|---|---|
| 4 | 8 | 43 | 3 | 930 | 2 |
| 4 | 16 | 60 | 3 | 999 | 7 |
| 8 | 4 | 40 | 4 | 942 | 3 |
| 8 | 16 | 81 | 3 | 1060 | 5 |
| 16 | 4 | 58 | 3 | 1003 | 8 |
| 16 | 8 | 82 | 3 | 1080 | 7 |

Table 2.4 shows the execution times when the system scales up and down. The overhead of the system increases with the increasing number of the compute nodes. The short computation time, the nodes that need to be configured and the data redistribution result in extra overhead with large number of compute instances. However, the overheads are still quite low.

## 2.5 Summary

This chapter has described the initial work towards the goal of making existing MPI applications elastic for a cloud framework. A simple illustration of the overall approach has been proposed and several runtime modules have been developed. This approach is based on terminating one execution and starting another with a different number of nodes. It is a static model based on a fixed iteration time where the monitoring is done withing the application itself. The evaluation with 2 applications has shown that this has small overheads, and elastic execution of MPI programs in cloud environments is feasible.

The following chapter gives a detailed account of the design and implementation of the decision layer and its ability to meet the constraints specified by the userusing a dynamic feedback model.

# CHAPTER 3

# DECISION LAYER

The first part of the thesis focuses on the design of the overall structure of the framework and gives a simple depiction of how the elasticity can be made possible by making a small modification to the source code. The control flow and the execution flow have been discussed in detail. The various components including check pointing, resource allocator, redistribution and restart and monitoring layer along with their functions have been described elaborately. The framework has been evaluated with a couple of applications to show that the overhead of redistribution and restart are negligible as compared to the overall execution time.

This chapter shows how the framework has been extended and modified to meet the user constraints of time and cost, with the help of a feedback model. It shows how the decision layer can completely handle the monitoring of the application by communicating frequently with the it and using the inputs fed by it. There are several factors that should be taken into consideration while modeling the decision layer. The user has the flexibility of choosing between the maximum time that the application can take to complete, or the maximum cost that he can afford to spend. The decision layer is the uppermost layer that decides the number and type of EC2 instances that are required to be deployed to meet these constraints.

The following section gives a detailed account of the decision layer design and implementation, which is followed by the experimental section showing the results and comparison of experiments conducted on two different applications. Section 3.5 summarizes the work.

## 3.1 Decision Layer Design

The main goals of the decision layer is to meet the user demands. As mentioned before, these constraints being *soft* and not *hard*, the decision layer, on recieving a maximum time or maximum cost input from user, will make best efforts with the available resources and components to complete the execution of the application within the specifications given. The term Best Effort here means that there is no guarantee that the execution will be completed within the given time or cost. The time taken per iteration, which is communicated by the application to the decision layer, is used to determine the progress of the application. In addition to this, the communication time between nodes is also taken into consideration, as an increase in the communication time affects scalability to a large extent. For communication intensive applications, it is obvious that an increase in the number of nodes will not improve the processing time. So rather than scaling up the number of nodes, the decision layer shifts to large instances on EC2, which have higher input/output as compared to the small instances, hence speeding up the communication. When time is specified as a criteria, the framework makes an attempt to finish within the given time, minimizing the cost as much as possible for the given time input. Similarly, when cost is specified as criteria, the framework makes its best attempt to finish within the budget, with the best processing time possible.

The decision layer has to interact with the application at regular intervals to monitor it and determine its progress. Hence there has to be some kind of communication between the two layers. TCP is used for message communication between the decision and the application layer. During the monitoring interval (which is specified by the decision layer), the application layer informs the decision layer of the average time taken for each iteration till that point. The latter makes a decision with its feedback model and informs the application if it has to continue or change. In the case of a decision being made to change, the data (live variables and matrices) should be collected and redistributed to the new set of nodes. The reason for using TCP is that the overhead of communication using TCP is very small and negligible. In addition, TCP also provides a reliable, ordered delivery of a stream of bytes from one program to the other.

There are a plethora of factors to be considered while designing the decision making model. Some of the primary factors include total input time / cost, current progress of the application, total number of iterations in the application, number of iterations completed, present node count, communication time per iteration and the overhead costs, which include the extra time taken for restart, redistribution and file read of the data set. A feedback model is used and the above parameters are given as input to the model. The application is started on a particular number of nodes, which can be varied. Typically , a minimum number of instances is used at start and depending on the progress, this number is scaled.

## 3.2 The Feedback Model

The model initially designed was a simple one with no feedback. It was targeted on achieving a fixed iteration time , which was calculated using the total input time and the number of iterations. The required iteration time is given by equation (3.1).

Table 3.1: Legend

| Symbol | Meaning |
|---|---|
| $t_{\mathrm{rit}}$ | Required Iteration Time |
| $t_{\mathrm{c}}$ | Time Constraint |
| $t_{\mathrm{oh}}$ | Overhead Time |
| $it_{\mathrm{tot}}$ | Total Number Of Iterations |
| $it_{\mathrm{rem}}$ | Remaining Number Of Iterations |
| $n_{\mathrm{c}}$ | Current Node Count |
| $n_{\mathrm{p}}$ | Predicted Node Count |
| $t_{\mathrm{tn}}$ | Time Taken Till Now |
| $t_{\mathrm{est}}$ | Estimated Time Using Current Nodes |
| $c_{\mathrm{tn}}$ | Cost Taken Till Now |
| $c_{\mathrm{pn}}$ | Cost Per Node Per hour |
| $c_{\mathrm{c}}$ | Cost Constraint |
| $bc$ | Billing cycle |

$$t_{rit} = \frac{t_{ip}}{it_{tot}} \tag{3.1}$$

After every monitoring interval, the current iteration time was checked to see if it was close to the above value. The number of nodes was decreased on account of the current iteration time being lesser than the required iteration time. If vice versa was true, then the number of nodes was increased. This model, being a static one, did not take the present state of the network into consideration. Since performance

on the cloud vary from time to time, this model was replaced by a feedback model which was more stable and made dynamic estimations rather than a static one.

### 3.2.1 Time Constraint

The time taken for one iteration on one node (Assuming perfect scalability) is given by

$$n_c \times t_{pi} \tag{3.2}$$

Time taken for remaining iterations on new number of nodes is given by

$$\frac{n_c t_{pi} it_{rem}}{n_n} \tag{3.3}$$

Time taken for remaining iterations on new number of nodes should ideally be

$$t_c - t_{tn} \tag{3.4}$$

By equating (3.3) and (3.4) , the new number of nodes can be found out as

$$n_n = \frac{n_c t_{pi} it_{rem}}{t_c - t_{tn}} \tag{3.5}$$

A measure of communication time per iteration is also made during each monitoring interval. If this value contributes to more than 30% of the total iteration time, the application is run on large instances which have higher input output performance as compared to the small instances . This will help reducing the time taken for communication.

### 3.2.2 Cost Constraint

The main factors that determine cost are the number of instances active, the time for which they are active, and the billing cycle. The billing cycle term here, refers

to the period between billing of the instances. The instance count is generally not changed when in the middle of a billing cycle, as termination of instances is still going to result in those instances being charged for the entire billing cycle. The decision layer thus waits till the billing cycle is completed, before making decisions. It is not possible to consider perfect scalability here, since a perfectly scalable model will result in decreasing time with increase in the number of instances. For example, if the number of instances is doubled, the time will be halved. So in both the cases the total cost spent will be the same. However since real life applications do not have perfect scalability, certain predictions can be made.

The communication time is observed and if it is too high, it is obvious that scaling up will not help reducing the processing time to a great extent. In this case, the number of instances is not increased, but a decision might be made to shift to large instances, depending on whether they will meet the cost constraints or not. If the communication cost is not so high, a node count which will meet the current cost constraints is predicted using equation(3.6). A check is then done by comparing the predicted time using the predicted nodecount and the estimated time using the current node count. If the current configuration is better, then the application is continued without termination. Otherwise it is terminated and restarted on the new number of instances.

$$n_p = \frac{bc(c_c - c_{tn})}{it_{rem} t_{pi} c_{pn}} \tag{3.6}$$

$$t_p = t_{tn} + \frac{it_{rem} t_{pi} n_c}{n_p} \tag{3.7}$$

$$t_{est} = t_{tn} + it_{rem}t_{pi} \tag{3.8}$$

Equation(3.7) and Equation(3.8) give the predicted time using the predicted node count and estimated time using current node count respectively.

### 3.2.3 Additional Factors

In addition to the above model, there are some additional factors that should be taken into consideration to make the decision layer more stable and reliable. Experiments were conducted using the above model with the two applications - jacobi and cg. It was observed that the results were not very close to the desired ones and the feedback model was not accurate. When scaling of nodes is done, in addition to processing of application, extra time was taken for restart, redistribution and file read and write. There was some loss of accuracy and frequent node changes as these costs were not considered. The results of this inaccurate model is listed in the experimental section. Hence the model was modified by taking the overhead time into consideration. Equation(3.5) and equation(3.7) will thus be modified as follows.

$$n_n = \frac{n_c t_{pi} it_{rem}}{t_c - t_{tn} - t_{oh}} \tag{3.9}$$

$$t_p = t_{tn} + t_{oh} + \frac{it_{rem}t_{pi}n_c}{n_p} \tag{3.10}$$

Equation(3.9) and Equation(3.10) show the modified versions with the overhead time taken into consideration.

The monitoring interval is another factor which should be given an optimum value. If monitoring is done too frequently, it might result in unnecessary overhead. If it is

done occasionally, it might result in loss of accuracy and failure to scale up at the right time to meet the constraints. There is an option of varying monitoring interval as well, where the monitoring interval is increased everytime a decision is made to continue on the same number of nodes. Experiments were conducted with each of these different cases and an optimum value of monitoring interval was chosen.

There might be cases where the user constraints provided might not be realizable ; for instance, the time or cost constraint might be so small, that it might not be possible to finish within the given budget or time. In such situations, the decision layer does its best by using maximum instances when the criteria is time, and minimum instances when the criteria is cost.

## 3.3 Implementation

The decision layer interacts with the user to obtain user inputs that include the application name, time or cost criteria, the initial node count on which the application needs to be started on and the initial monitoring interval. The resource allocator allocates the necessary resources and also takes care of the MPI setup and configuration before every restart of the application. MPICH2 hydra process manager is used in this part of the work as its setup and configuration is much simpler compared to that of the mpd process manager.

Fig. 3.3 shows the outline of the decision layer. The inputs are taken from the user and the output includes total process time or total cost depending on the criteria chosen. The node count is initialized to the initial node count given by the user. The resource allocator module is called with the current node count for allocation of the necessary resources. The application source file is copied to all the instances using

scp command and this is done using parallel threads to reduce overhead. Module scpDtoM() does a copy of the source code from the Decision layer to the other worker nodes. If the loopcount is greater than 1 (if the application is not starting for the first time, but rather restarting due to a node change decision), the scpDtoM module is used to redistribute the matrix portions to all the nodes. The application is then launched with run time arguments that include monitoring interval, the hostname of the local machine and the tcp port number which are used for tcp socket communication. The application launcher module is used for this. The decision layer waits on a receive till a particular number of iterations is over and the iteration time, communication time, the iteration number and a message is sent by the master node which is running the MPI application. If the message is read as 'finished', it signifies the completion of the application and the while loop terminates. If a finished message is not received, then the decision layer with the received inputs - the iteration time and the communication time, makes a decision using the feedback model and sends an instruction to the Master node to either continue or change. On receiving continue, the application continues running until the next monitoring interval and on receiving change, the application outputs its live matrices either to a file or to an Amazon S3 bucket. These are collected at the decision layer which combines all of them and re-splits them using combineandsplit module. This process repeats itself until the application completes.

**I**nput: *monitor_iter*,*criteria*, *init_nodecount*
*time_criteria* or *cost_criteria*
**O**utput:*process_time*, *process_cost*

```
complete := False;
nodecount := init_nodecount;
loopcount := 1;
While complete ≠ True Do
   Call resourceallocator(nodecount);
   Call scpDtoM(filename);
   If loopcount>1 Then
        Call scpDtoM(portions);
   Endif
   Call applicationlauncher(nodecount);
   While True Do
        Call receive(message,itertime,commtime);
        If message =′ finished′ Then
             complete := True;
             break;
        Endif
        decision := feedbackmodel(itertime,commtime);
        Call send(decision);
        If decision =′ change′ Then
             break;
        Endif
   Endwhile
   If complete ≠ True Then
        Call scpMtoD(active);
        Call combineandsplit();
   Endif
   loopcount := loopcount+1;
Endwhile
```

Figure 3.1: Execution Structure : Decision layer

## 3.4 Experimental Results

In this section, the decision layer is evaluated and the performance is measured for different values of time and cost criteria. The same applications - jacobi and cg were used for the experiments. The dataset of jacobi consists of a matrix consisting of (5K $\times$ 5K) double values. The solution converges in about 1000 iterations and it is a compute intensive application. The dataset of CG is a sparse matrix of (150K $\times$ 150K) and the total number of iterations is about 75 iterations. CG is a communication intensive application.

### 3.4.1 Criteria - Time

The time criteria was chosen and experiments were conducted with different values of time criteria in seconds. The decision layer starts the application and monitors it and uses the feedback model to figure out if a restart is necessary. The total processing times (Actual Time) were recorded to see how close they are to the time criteria and observations were made.

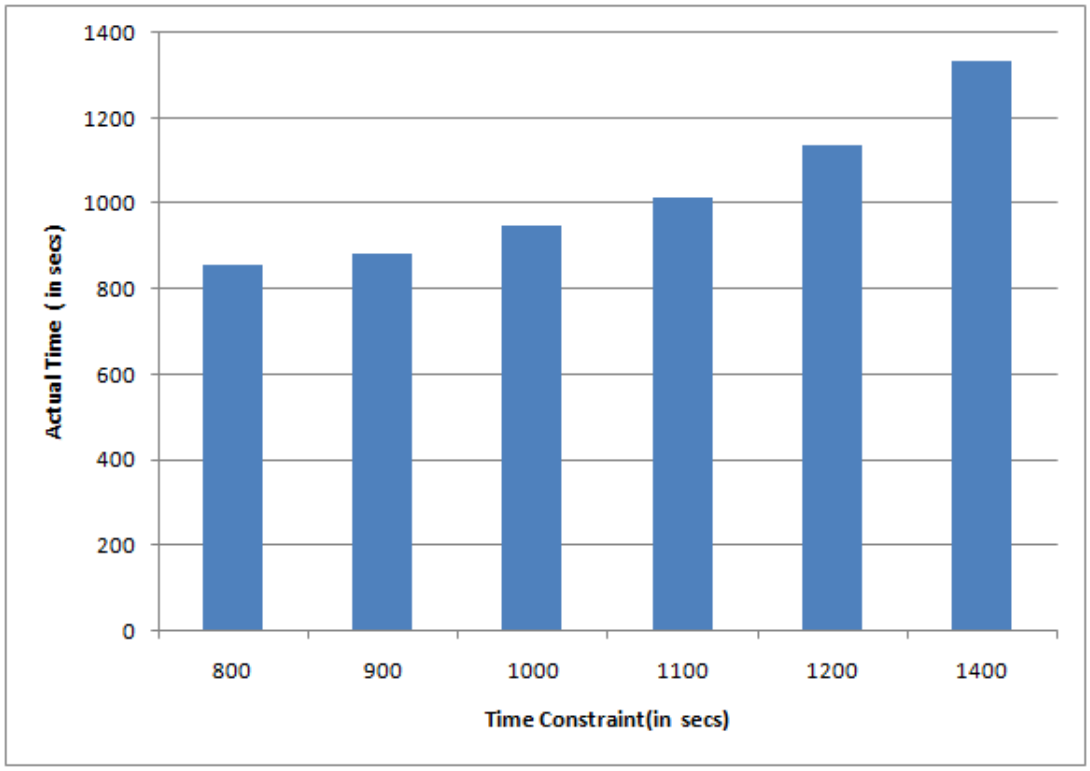


Figure 3.2: Time Criteria vs Actual Time - Jacobi

### Jacobi

Figure 3.2 shows the different values of total process times for various time criteria in seconds. It is observed that the actual time taken is lesser than the time for all values other than 800 seconds. In this case, the system is not able to meet the requirements even with maximum number of nodes. Hence it uses the maximum number of resources and makes a best effort to give a good performance. For a time criteria of 800 seconds, the actual time is 875 seconds. For all other values of time criteria, the system keeps the actual time within the specified limit, at the same time minimizing the cost by using an optimum number of resources

Figure 3.3 shows the final node count values for different values of time criteria. Although each of these runs start at a specified initial value of node count, they converge to an optimum value after a certain point and remain there. With the increase in maximum specified user time, the system uses fewer number of resources as it can finish within the specified time and still reduce the cost as much as possible.

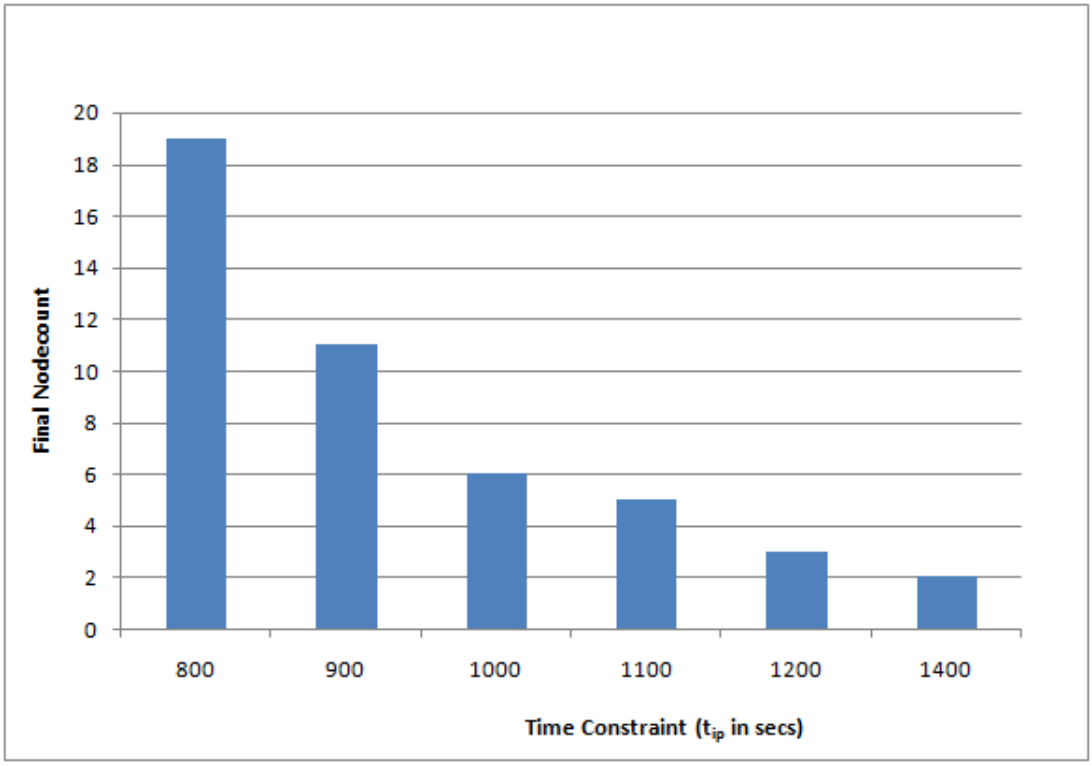


Figure 3.3: Time Criteria vs Nodecount - Jacobi

It is observed that the final node count value decreases with increase in time criteria. For a time criteria of 800 secs, the system uses 19 nodes , which is the maximum in this case. For a higher time criteria, the decision layer allocates fewer

nodes so that the cost is minimized as much as possible, at the same time keeping the total process time within the given limit.

Figure 3.4 shows the node changes for different time criteria.

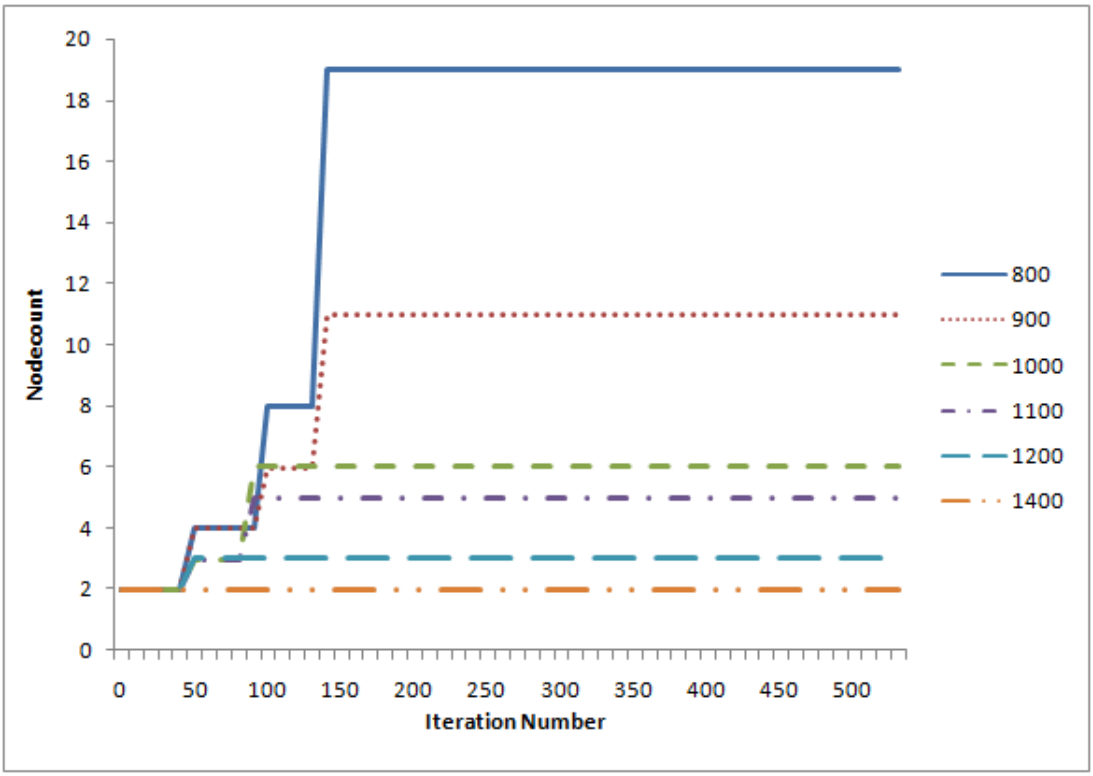


Figure 3.4: Nodechanges for different Time Criteria - Jacobi

All the node changes are within 140 iterations of the 1000 total iterations, after which it stabilizes to an optimum value for the remaining iterations. This shows the decision layer is able to make the right decision quickly without restarting too many number of times.

Figures 3.5, 3.5 and 3.7 compare the two different versions of the feedback model. The first one was where the overhead of restart and redistribution were not given as inputs to the feedback model. The second version was one which included this overhead. These comparisons are shown for three different values of input time.

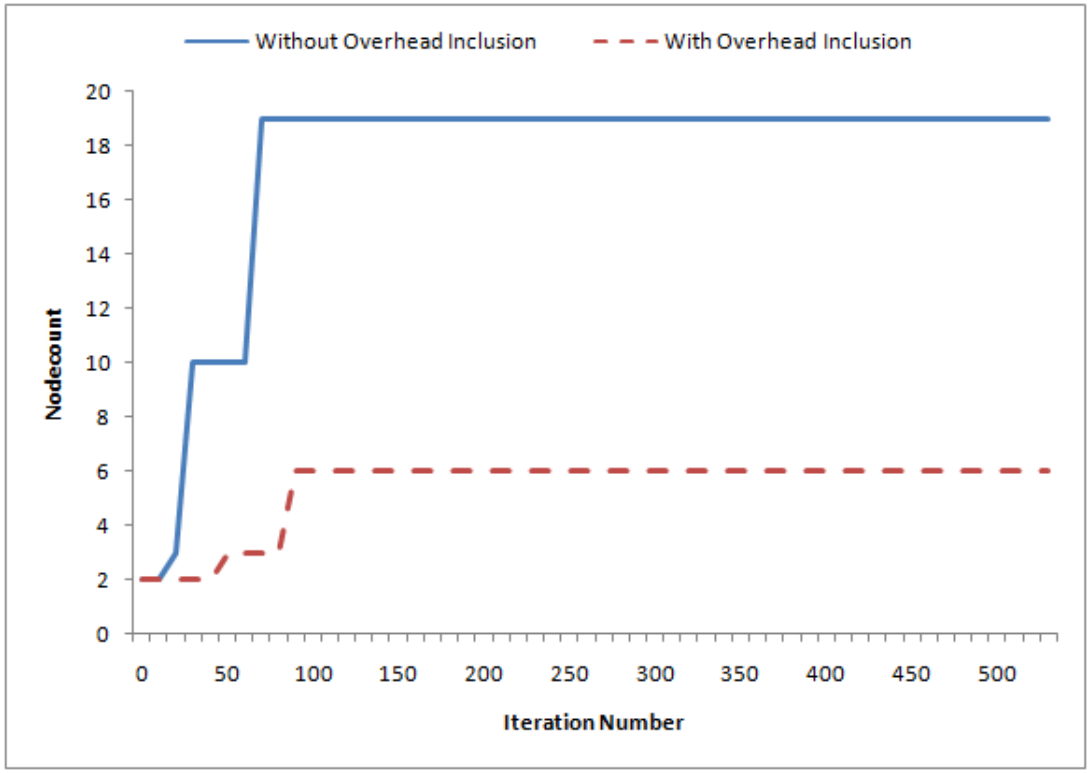


Figure 3.5: Comparison of different versions for Time Criteria=1000 secs - Jacobi

It is observed that the second version is able to make a better decision as compared to the first one. In the first version, since the overhead is not taken into account, even after restart the system does not behave as it predicted. Hence restart occurs again and finally reaches the maximum node count in every case.

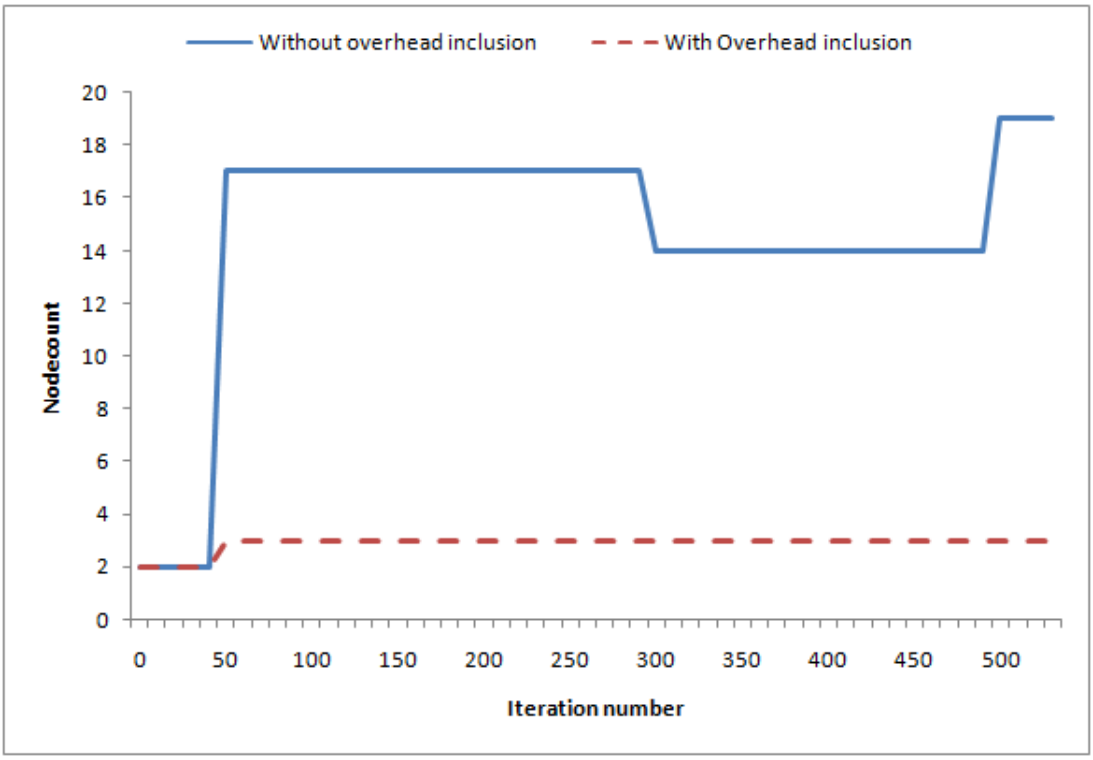


Figure 3.6: Comparison of different versions for Time Criteria=1200 secs - Jacobi

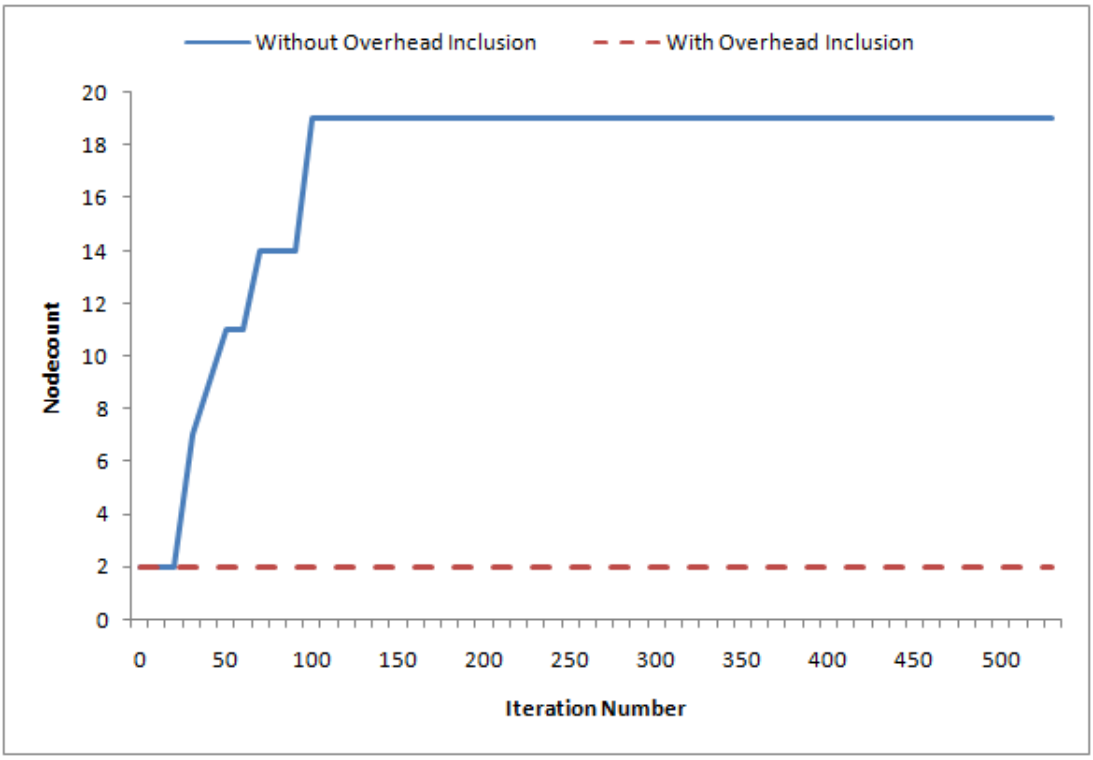


Figure 3.7: Comparison of different versions for Time Criteria=1400 secs - Jacobi

**Conjugate Gradient (CG)**

Similar experiments were repeated with the CG application and interesting observations were made. Since CG is a communication intensive application, it behaves very differenly from that of Jacobi. With different values of start nodes, Table 3.2 shows how node changes occur for different values of input time.

Table 3.2: Node Changes for different Time Criteria and start node types-CG

| **Start Node** | **4 small** | | | **4 large** | | |
|---|---|---|---|---|---|---|
| Time Criteria(sec) | Nodechange | Iteration Number | Actual Time(sec) | Nodechange | Iteration Number | Actual Time(sec) |
| 300 | 4 small to 4 large | 5 | 448 | remains at 4 large | - | 357 |
| 600 | 4 small to 4 large | 35 | 592 | remains at 4 large | - | 371 |
| 800 | stays at 4 small | - | 764 | 4 large to 4 small | 5 | 690 |

The experiments were conducted with different start node types - small and large. It was observed that all the experiments with the same time criteria ,converged to the same instance type at the end. When a small time criteria is given , changing to large instance will speed up the application to a large extent, since the communication time of this application is very high. On the contrary, when a large time criteria is given , the system continues to run the application on small instances as they would turn out to be less expensive than the large ones, hence reducing the overall cost, at the same time meeting the time constraint.

### 3.4.2 Criteria - Cost

This section shows the experimental results for cost criteria in jacobi and cg. The second criteria - cost was chosen and the experiment was run for different values for cost criteria for which the actual cost, time and nodechanges were recorded.

**Jacobi**

Figure 3.8 gives the actual cost for different values of cost criteria, for jacobi application.

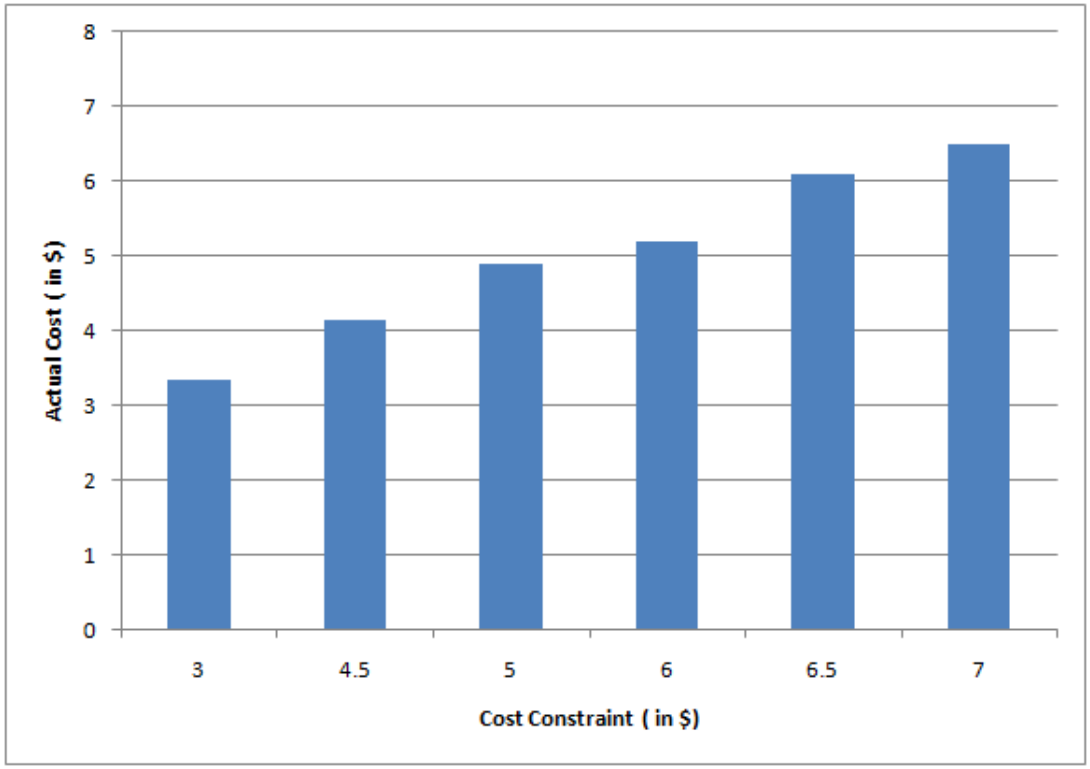


Figure 3.8: Cost Criteria vs Actual Cost - Jacobi

The observation is similar to the time section of jacobi, as the demand is met for all values of cost criteria except the first one , i.e 3$. When this input is given, the system does its best by trying to minimize the cost as much as possible by using minimum resources. For all values, an optimum number of resources are used, hence taking the time also into consideration. The more the user can afford to spend, the better performance he will get out of it. This is seen in Figure 3.9, where the output processing time improves as the cost criteria increases.

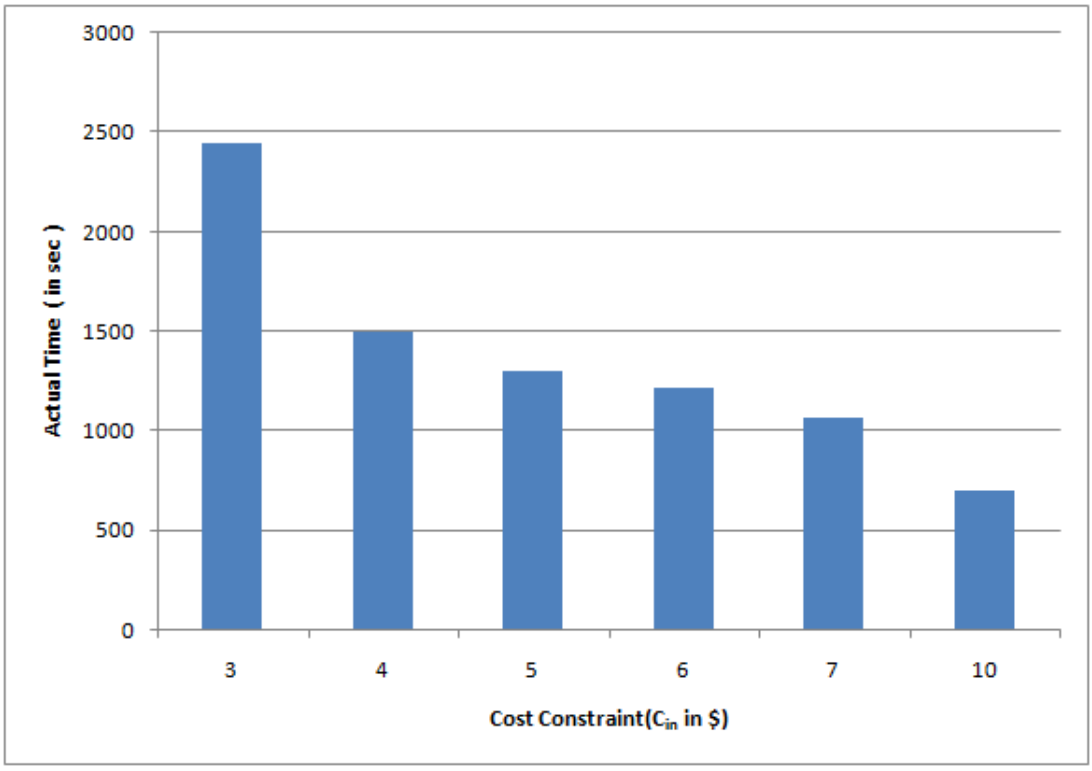


Figure 3.9: Cost Criteria vs Actual Time - Jacobi

Figure 3.10 shows the final node count value for different values of cost criteria.

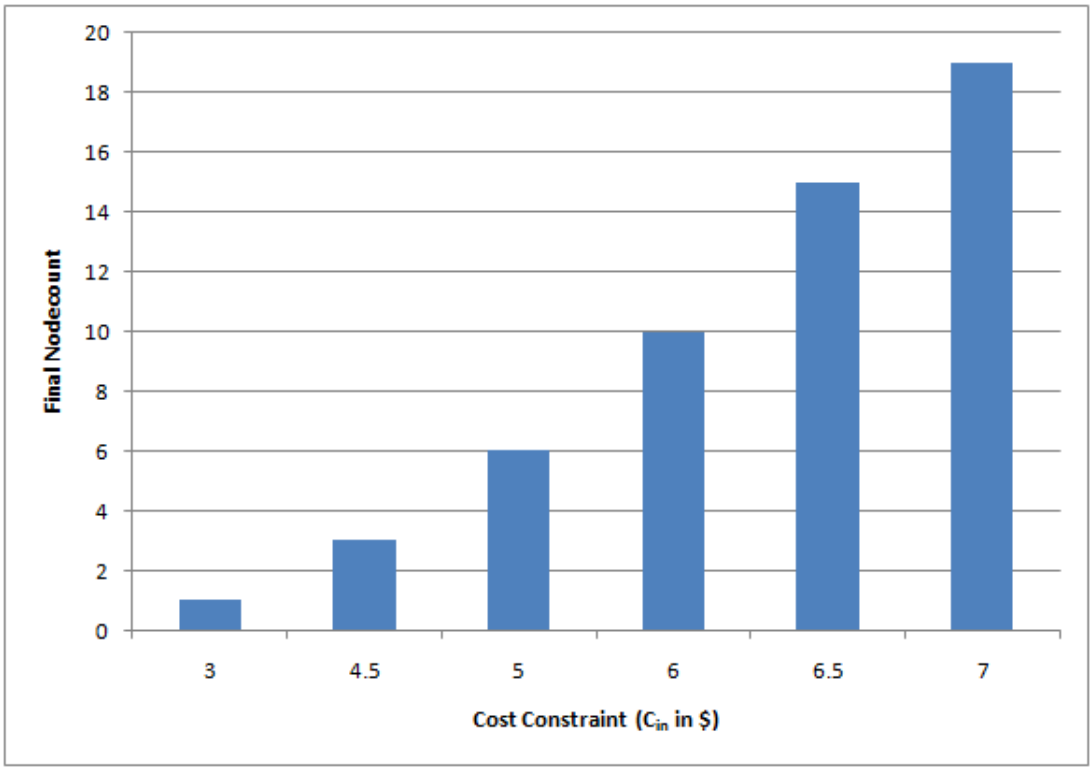


Figure 3.10: Cost Criteria vs Final Nodecount - Jacobi

As cost increases, more number of resouces are allocated to complete the application within lesser time.

Figure 3.11 shows the nodechanges for different values of cost criteria. Similar to the

time criteria, the decision layer is able to stabilize and optimize the resources, while satisfying the cost criteria

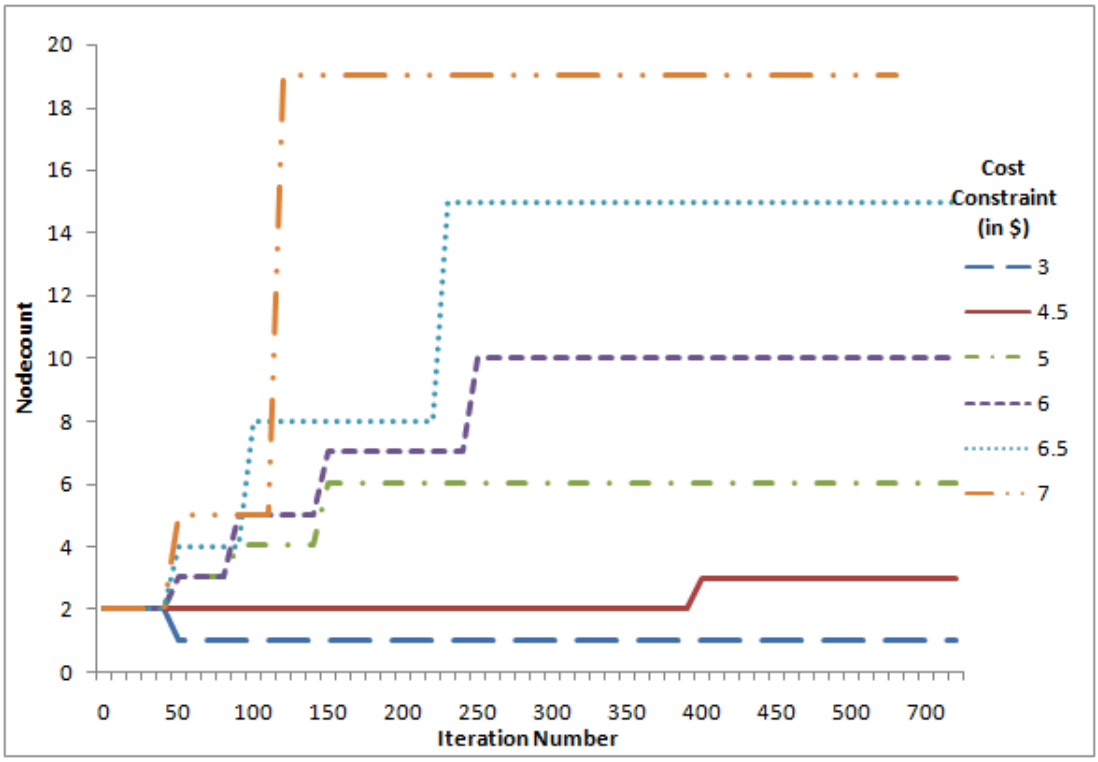


Figure 3.11: Nodechanges for Different values of Cost Criteria - Jacobi

### Conjugate Gradient (CG)

Figure 3.12 shows the actual cost for different values of cost criteria in CG.

Simlar to jacobi, we observe that the actual cost is very close to the cost criteria. Figure 3.13 shows the actual time for different values of cost criteria. The process time improves and gets better as the cost increases.

Figure 3.3 shows the node changes for different cost criteria. When the cost is really low, smaller nodes are used to minimize the cost. In other cases, the nodechange

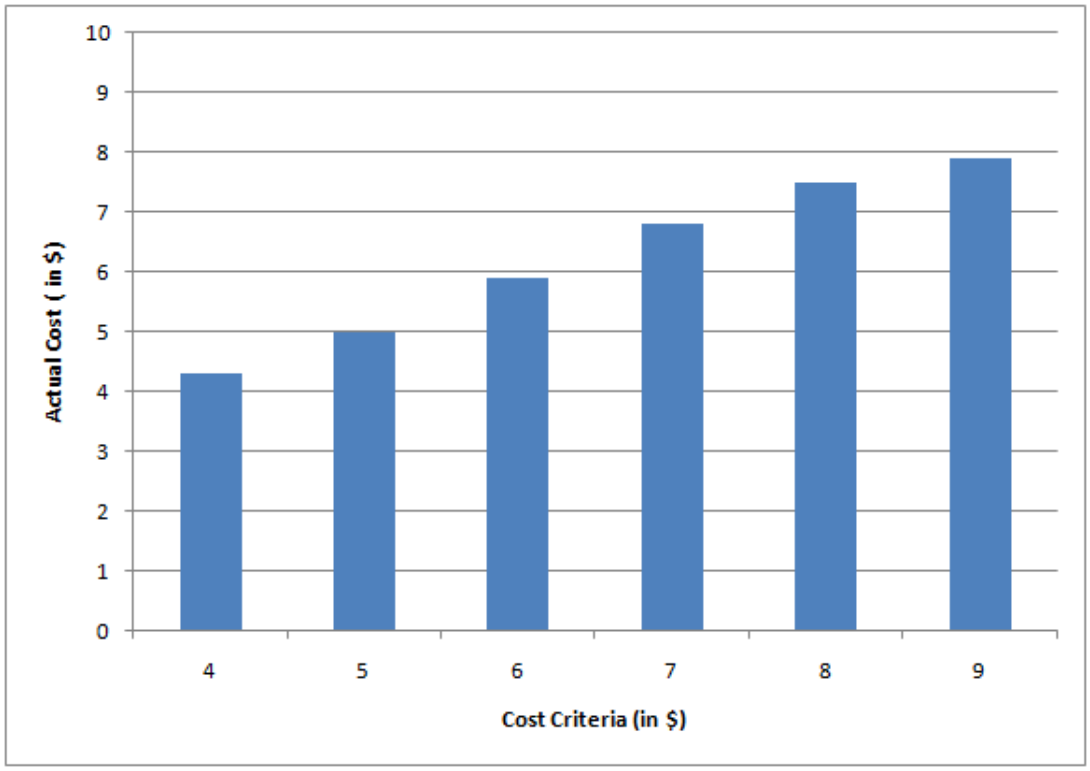


Figure 3.12: Cost Criteria vs Actual Cost - CG

from small to large occur at different iterations to optimize between time and total cost.

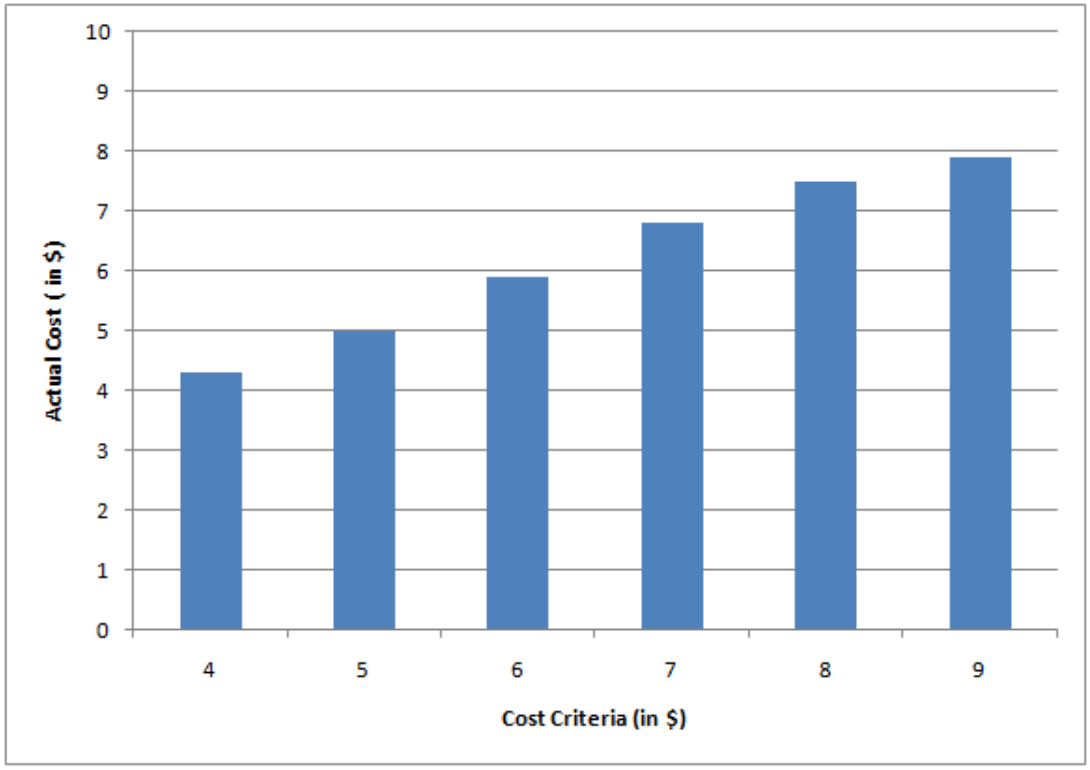


Figure 3.13: Cost Criteria vs Actual Time - CG

Table 3.3: Node Changes for different Cost Criteria - CG

| Cost Criteria ($) | Iter. No | Nodechange |
|---|---|---|
| 4 | Never | 4 small |
| 5 | 45 | 4 small to 4 large |
| 6 | 40 | 4 small to 4 large |
| 7 | 35 | 4 small to 4 large |
| 8 | 25 | 4 small to 4 large |
| 9 | 5 | 4 small to 4 large |

## 3.5 Summary

In this work, the feedback model is used for determining the progress and type of application, whether it is a compute or communication intensive application. This information is necessary for finding out the number of resources that are to be allocated and the type of instances (large or small) that should be used. The decision layer meets the user demands of time or cost by making use of this feedback model.

A compute intensive application (jacobi) and a communication intensive application (CG) were used to evaluate the system. The results show that the decision layer makes best effort in meeting the user demands. The pattern of the node changes also shows that the number of instances stabilizes after certain number iterations, avoiding excessive overhead of node change.

# CHAPTER 4

# CONCLUSIONS

In this work, we have proposed a method for scientific applications to exploit the elasticity of the cloud environments. MPI applications are designed for a fixed number of nodes. We have designed a framework that can make existing MPI applications elastic and adaptable using an automated framework which decides the number of instances for execution based on user demands of time or cost.

The Framework is tested using 2 MPI applications , one of them is communication intensive and the other one is compute intensive. The experiments show that the overheads during elastic execution are low and tolerable. Is is also shown that ths system makes best efforts to meet the given user constraints of time or budget.

# BIBLIOGRAPHY


[1] David Chiu, Apeksha Shetty, and Gagan Agrawal. Elastic cloud caches for accelerating service-oriented computations. In *Proceedings of SC*, 2010.

[2] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. ElasTraS: An Elastic Transactional Data Store in the Cloud. In *Proceedings of Workshop on Hot Topics in Cloud (HotCloud)*, 2009.

[3] Ewa Deelman, Gurmeet Singh, Miron Livny, Bruce Berriman, and John Good. The cost of doing science on the cloud: the montage example. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[4] Guy Edjlali, Gagan Agrawal, Alan Sussman, and Joel Saltz. Data parallel programming in an adaptive environment. In *Proceedings of the Ninth International Parallel Processing Symposium*, pages 827–832. IEEE Computer Society Press, April 1995.

[5] Chao Huang, Gengbin Zheng, Laxmikant V. Kalé, and Sameer Kumar. Performance evaluation of adaptive mpi. In Josep Torrellas and Siddhartha Chatterjee, editors, *PPOPP*, pages 12–21. ACM, 2006.

[6] Gideon Juve, Ewa Deelman, Karan Vahi, Gaurang Mehta, Bruce Berriman, Benjamin P. Berman, and Philip Maechling. Scientific workflow applications on amazon ec2. *CoRR*, abs/1005.2718, 2010.

[7] Derrick Kondo, Bahman Javadi, Paul Malecot, Franck Cappello, and David P. Anderson. Cost-benefit analysis of cloud computing versus desktop grids. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.

[8] Harold Lim, Shivnath Babu, and Jeffrey Chase. Automated Control for Elastic Storage. In *Proceedings of International Conference on Autonomic Computing (ICAC)*, June 2010.

[9] Jie Li, *et al.* escience in the cloud: A modis satellite data reprojection and reduction pipeline in the windows azure platform. In *IPDPS '10: Proceedings of the 2010 IEEE International Symposium on Parallel&Distributed Processing*, Washington, DC, USA, 2010. IEEE Computer Society.

[10] Ming Mao, Jie Li, and Marty Humphrey. Cloud Auto-Scaling with Deadline and Budget Constraints. In *Proceedings of GRID 2010*, October 2010.

[11] Daniel Nurmi, Richard Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. In Franck Cappello, Cho-Li Wang, and Rajkumar Buyya, editors, *CCGRID*, pages 124–131. IEEE Computer Society, 2009.

[12] Mayur R. Palankar, Adriana Iamnitchi, Matei Ripeanu, and Simson Garfinkel. Amazon s3 for science grids: a viable solution? In *DADC '08: Proceedings of the 2008 international workshop on Data-aware distributed computing*, pages 55–64, New York, NY, USA, 2008. ACM.

[13] Luis Rodero-Merino, Luis Miguel Vaquero Gonzalez, Victor Gil, Fermín Galán, Javier Fontán, Rubén S. Montero, and Ignacio Martín Llorente. From infrastructure delivery to service management in clouds. *Future Generation Comp. Syst.*, 26(8):1226–1240, 2010.

[14] R. Thakur, A. Choudhary, and J. Ramanujam. Efficient algorithms for array redistribution. *IEEE Transactions on Parallel and Distributed Systems*, 7(6):587–594, June 1996.

[15] Christian Vecchiola, Suraj Pandey, and Rajkumar Buyya. High-performance cloud computing: A view of scientific applications. *Parallel Architectures, Algorithms, and Networks, International Symposium on*, 0:4–16, 2009.

[16] Jon Weissman and Siddharth Ramakrishnan. Using proxies to accelerate cloud applications. In *Proc. of the Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[17] Jon B. Weissman. Predicting the cost and benefit of adapting data parallel applications in clusters. *J. Parallel Distrib. Comput.*, 62(8):1248–1271, 2002.